Data-driven Recurrent Set Learning for Non-termination Analysis

***Abstract*—Termination is a fundamental liveness property for program verification. In this paper, we revisit the problem of non-termination analysis and propose the first black-box learning algorithm for synthesizing recurrent sets, where the non-terminating samples are effectively speculated by a novel method. To ensure convergence of learning, we develop a learning algorithm which is guaranteed to converge to a valid recurrent set if one exists, and thus establish its relative completeness. The methods are implemented in a prototype tool and experimental results on public benchmarks show its efficacy in proving non-termination as it outperforms state-of-the-art tools, both in terms of cases solved and performance. Evaluation on non-linear programs also demonstrates its ability to handle complex programs.**

***Index Terms*—program termination, recurrent set, data-driven approach, black-box learning**

I. Introduction

Proving termination is essential for establishing total correctness of programs. During the last two decades, automated termination analysis has received abundant research, as practical tools like AProVE [1], CPAchecker [2], Ultimate [3] and 2LS [4] emerge, which are capable of analyzing complex programs.

Since termination is generally undecidable, proving non-termination, as the counterpart of proving termination, has also been an active research topic, and a plethora of methods for non-termination analysis exist. Typically, proving non-termination is more challenging because the witness to non-termination is an infinite execution trace. Similar to ranking function in termination analysis, the key is to find a symbolic representation that encapsulates requirements for non-termination, and thus serves as non-termination arguments. Most of the existing work on non-termination analysis is based on the notion of recurrent set [5], and methods have been proposed to synthesize it, most of them utilizing white-box template-based constraint solving (e.g. [6], [7], [8]).

A recent line of work on termination analysis is based on black-box learning, which has been successfully applied in invariant generation [9], [10], [11]. Some researches follow the method of counter-example guided inductive synthesis (CEGIS) for exact learning of machine learning models such as linear regression [12], decision tree [13] and support vector machine [14], [15]. Deep learning can also be utilized to train a neuron network as the termination argument [16], [17]. Since the synthesis process is driven by data collected from program, such methods are also called *data-driven* approaches.

Generally, in black-box learning, the synthesizer contains two main components. One of them is a *learner*, which tries to produce a candidate using *samples* or *data points* obtained from the program. Another is a *teacher*, which acts as a verifier that validates a conjectured model the learner generates. In case the candidate is invalid, the teacher generates new samples from the counterexample, or by using techniques like bounded model checking and dynamic program execution. The framework is depicted in Figure 1.

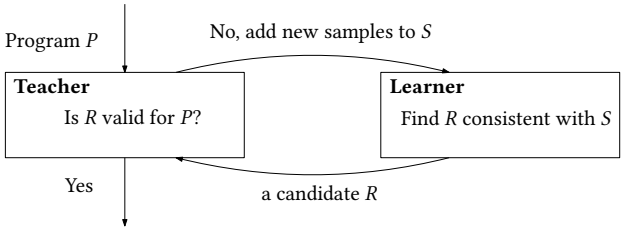


Fig. 1: The typical black-box learning framework

Compared with white-box approaches, the design of a black-box synthesizer is simpler. Besides, black-box learning enjoys the advantage that the learner is completely agnostic of the concrete program and its semantics. Thus, they are able to reason about programs with constructs that are otherwise hard to deal with, such as non-linear assignments. Many white-box approaches also proceed by analyzing *lasso*-shaped traces (i.e. a finite *stem* and a periodic *loop*) extracted from the program [5], [7]. This is problematic in non-termination analysis since aperiodic non-termination can not be detected. Black-box approach, however, does not suffer this problem.

These inspire us to adopt black-box learning to non-termination analysis, as recurrent sets can be seen as predicates that are analogous to inductive invariants. However, generating non-terminating samples for the learner is difficult, since the execution trace starting from a non-terminating state is infinite. Such samples can not be obtained by dynamic execution or validity check of candidate predicates. Besides, finding a non-terminating state is sufficient for proving non-termination, which makes the traditional black-box method conceptually unnecessary. Therefore, to apply black-box learning for proving non-termination, the problem of how to generate these samples must be addressed.

In this paper, we propose the first algorithm for proving non-termination by learning a recurrent set. As decision tree has been a well-studied model and optimal algorithms exist, the algorithm utilizes the decision tree learner to learn a candidate recurrent set, which is based on the ICE learning framework [10]. The samples are generated by static methods,

and are labeled either *positive, negative*, or *implicative*. The aforementioned problem is tackled by speculation of positive samples from reachable states using a novel method based on constraint solving. To reduce the penalty of selecting a spurious sample, we propose to utilize the unsat core to generalize the positive samples.

The convergence of learning is important for black-box approaches. We demonstrate a variant of bounded learning algorithm that is guaranteed to terminate, provided that a recurrent set representable as a decision tree exists (in a sense described in Section II-B). This thus ensures the relative completeness of our method. As far as we know, there is no established non-termination proving technique that is based on black-box learning and provides such guarantee.

We have implemented our algorithm in a prototype tool called RSLEARN. Experimental results on public benchmarks show the efficacy of our method as it proves more non-terminating cases than state-of-the-art termination and non-termination analysis tools such as ULTIMATE [3], VERY-MAX [18] and REVTERM [8] within fewer time. Besides, evaluation on non-linear benchmarks show its ability to handle non-linear programs efficiently.

In summary, we present in this paper:

- A black-box algorithm based on decision tree learning for synthesizing recurrent set and proving non-termination of programs that generates non-terminating samples with a novel speculation method.
- A bounded learning algorithm which guarantees convergence of the learning process as long as an expressible recurrent set exists.
- An implementation of the algorithm, and the experimental results of the algorithm on public benchmarks which show the effectiveness of the methods.

The rest of the paper is organized as follows: Section II introduces basic definitions and decision tree learning, and Section III demonstrates the algorithm by a motivating example. Section IV discusses the learning algorithm and Section V introduces the bounded convergent algorithm. Section VI describes the implementation and experimental results and Section VII briefly reviews related work. We conclude the paper in Section VIII.

II. PRELIMINARIES

A. Basic Definitions

Let $\vec{x}$ be the set of variables that occur in a program and $\vec{x}'$ be the set of primed version of variables. A *state* s is defined as a valuation of $\vec{x}$, and Σ is the set of all states. A *transition* is a pair of states (s, s'). Let $\sigma_{s,s'}$ be the combined valuation of s and s' such that for each $v \in \vec{x}$, $\sigma_{s,s'}(v) = s(v)$ and $\sigma_{s,s'}(v') = s'(v)$. A *transition formula* f has free variables ranging over $\vec{x} \cup \vec{x}'$, and denotes a set of transitions. The transition (s, s') satisfies f, written $(s, s') \models f$, if $\sigma_{s,s'} \models f$.

For brevity, we do not introduce a specific high-level programming language and its semantics, but instead adopt the standard techniques [19] to represent a loop as transition formulas as in Figure 2.

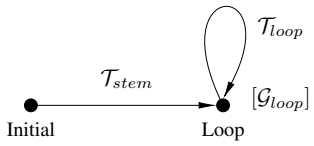


Fig. 2: A loop L represented as transition formulas

Definition 1 (Loop)**.** *A* loop L *is a triple* $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$*, where*

- $\mathcal{T}_{stem}$ *is a transition formula representing all transitions from the initial states to the loop entry,*
- $\mathcal{G}_{loop}$ *is a predicate representing the condition for entering the loop, and*
- $\mathcal{T}_{loop}$ *is a transition formula representing all transitions from the beginning of the loop body back to the loop.*

Programs can be transformed to transition formulas using established methods (e.g. algebraic program analysis [20]) that can handle multiple loops and nested loops. For example, the loop in Figure 3 can be represented as

$$L_0 = (\texttt{true},\ k \neq 0,\ k' = -2 \times (k-1) \times k \times k \wedge j' = 0).$$

A loop is said to be *non-terminating*, if there is an infinite sequence of states $s_0, s_1, s_2, \cdots$ such that $(s_0, s_1) \models \mathcal{T}_{stem}$ and for every $i \geq 1$, $s_i \models \mathcal{G}_{loop}$, $(s_i, s_{i+1}) \models \mathcal{T}_{loop}$. The canonical method for proving non-termination is to synthesize a so-called *recurrent set*. An *open recurrent set* [5] is a non-empty set of states such that for every state in the set, at least one of its successor state under $\mathcal{T}_{loop}$ is also in the set. As a result, once the program reaches a state in the open recurrent set, it has the possibility to stay in this set for ever. In contrast, a *closed recurrent set* [21] is a non-empty set of states such that every successor of its states is contained in it. Thus, once the program enters, it can never escape (for all executions).

It can be easily shown that a program is non-terminating iff the program admits an open recurrent set. However, synthesizing an open recurrent set requires to eliminate $\forall\exists$-quantifications, which is usually difficult and inefficient [21]. We thus consider the closed recurrent set in this paper.

Definition 2 (Closed Recurrent Set)**.** *A* closed recurrent set *of a loop* $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$ *is a predicate* R *that satisfies:*

1) $\varphi_1 : \exists \vec{x}, \vec{x}'.\ \mathcal{T}_{stem}(\vec{x}, \vec{x}') \wedge R(\vec{x}')$
2) $\varphi_2 : \forall \vec{x}.\ R(\vec{x}) \rightarrow \mathcal{G}_{loop}(\vec{x})$
3) $\varphi_3 : \forall \vec{x}, \vec{x}'.\ R(\vec{x}) \wedge \mathcal{T}_{loop}(\vec{x}, \vec{x}') \rightarrow R(\vec{x}')$

The predicate R can be regarded as a set of states. In the definition, the first condition φ_1 checks if the set R is reachable from a certain initial state, the second condition φ_2 ensures no final state is contained in R, and the third condition φ_3 ensures all successor states of R are in R (i.e. R is inductive). Each φ_i $(1 \leq i \leq 3)$ is called a *validation formula*. In the following, we write ϕ for $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ for simplicity.

Note that R in the above formulas is a placeholder. We employ a "guess-and-check" approach to infer the predicate R. Let H be a *candidate* predicate for R. $\varphi_1(H)$ denotes the formula obtained by replacing R with H in φ_1, and we can likewise define the same for $\varphi_2, \varphi_3, \phi$, etc. The validity of $\phi(H)$ can be checked by an SMT solver. If $\phi(H)$ is valid, we say the candidate predicate H represents a *valid* recurrent set; otherwise, the SMT solver returns a model that falsifies $\phi(H)$. Depending on the model falsifying $\varphi_1(H), \varphi_2(H)$ or $\varphi_3(H)$, we can extract one or two states from that model.

Closed recurrent set is a stronger notion of recurrent set, and as a non-termination argument, it is only complete for a general loop when combined with an under-approximation [21]. In later sections, we assume $\mathcal{T}_{loop}$ is deterministic, i.e. it satisfies $\forall \vec{x}, \vec{x}', \vec{x}''. \mathcal{T}_{loop}(\vec{x}, \vec{x}') \wedge \mathcal{T}_{loop}(\vec{x}, \vec{x}'') \rightarrow \vec{x}' = \vec{x}''$. For a deterministic loop, open and closed recurrent sets are equivalent, thus the closed recurrent set is complete. In our implementation, an under-approximation is selected by a simple heuristic (Section VI) to handle non-deterministic loop.

B. Decision Tree Learning

Decision tree is a commonly adopted model in machine learning, and is especially used in classification problem. The model is simple but well-studied, with many classical decision tree learning algorithm such as ID3 [22], C4.5 [23] and C5.0. It has been recently used for invariant generation [9], [10], [24] and termination analysis [13] and shows promising results. We base our algorithm on decision tree learning as well.

Denote a numerical expression over $\vec{x}$ as an *attribute*. For example, the octagonal domain includes attributes of the form $\pm x \pm y$. Let $A = \{a_1, a_2, \ldots, a_n\}$ be a finite set of attributes. An *atomic predicate* is of the form $a \leq c$, where $a \in A$ and c is a constant. A decision tree learner iteratively tries to pick an atomic predicate, and partitions a state subspace into two that satisfy $a \leq c$ and $\neg(a \leq c)$ respectively. A decision tree is then a combination of the subspaces. More formally, a *decision tree* is defined as a boolean combination of atomic predicates. A predicate is *expressible*, if it can be represented as a decision tree. Note that if we fix the set of attributes, the set of semantically different decision trees, represented as boolean combinations of the atomic predicates, is infinite.

Given a set of samples with binary label (*positive* and *negative*), the classical algorithms learn a decision tree by computing the information gain of an atomic predicate. These algorithms, however, are tailored for inexact learning, and try to avoid overfitting by pruning the tree. Besides, similar to an inductive invariant, Definition 2 requires a recurrent set to be inductive (φ_3). It has been shown binary labeled samples are insufficient for inductive invariant synthesis [9].

In this paper, we follow the ICE framework for learning a decision tree [10], in which a sample set consists of three kinds of samples labeled positive, negative and implicative, respectively. A *positive* (resp. *negative*) sample is a state s, and requires the candidate must (resp. must not) hold on s. An *implicative* sample is a pair of states (s_1, s_2) that requires if a candidate holds on s_1, it must also hold on s_2. We then denote a sample set as a triple $S = (S^+, S^-, S^\rightarrow)$, with $S^+, S^-, S^\rightarrow$ being the set of positive, negative and implicative samples, respectively. A sample set S is called *inconsistent*, if there is no predicate that satisfies all requirements of its samples.

The set $S^\rightarrow$ of implicative samples can be seen as a partial function on Σ. Let $f(S) = \{s \mid \exists s' \in S.\ (s', s) \in S^\rightarrow\}$ and $g(S) = \{s \mid \exists s' \in S.\ (s, s') \in S^\rightarrow\}$. The closure of S^+ and S^- under $S^\rightarrow$, denoted as $S^+_\rightarrow$ and $S^-_\rightarrow$, is the transitive closure of S^+ and S^- under f and g, respectively.

The ICE decision tree learner takes a sample set as input and returns a candidate decision tree which meet all the requirements imposed by the samples, the details of which can be found in [10]. The following lemma states the learner is guaranteed to return a candidate as long as the sample set is consistent.

Lemma 1 (Theorem 1 of [10])**.** *Given a consistent sample set S, the ICE decision tree learner, independent of how attributes are chosen, always terminates and produces a decision tree consistent with S.*

```
while (k ≠ 0) {
    j ← -2 × (k - 1) × k;
    k ← j × k;
    while (j ≠ 0) {
        if (j > 0)
            j ← j - 1;
        else
            j ← j + 1;
    }
}
```

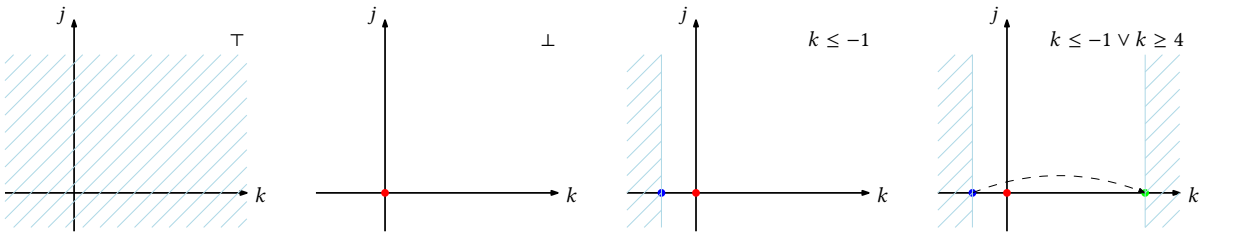
Fig. 3: A non-terminating motivating example

III. Motivating Example

For an overview of our algorithm, consider the program in Figure 3. This program does not terminate if the initial value of k is neither 0 nor 1, because k would keep flipping sign and get larger absolute value. The predicate $R = (k < 0) \vee (k > 1)$ characterizes a valid recurrent set of this program that proves its non-termination.

Though the program looks simple, it is difficult to prove its non-termination, since (1) this program is non-linear, so typical white-box synthesis methods, which are based on Farkas' Lemma, do not apply, (2) all valid recurrent set, including $(k < 0) \vee (k > 1)$, are disjunctive, and no conjunctive recurrent set exists, (3) every non-terminating trace of this program is aperiodic, meaning there's no lasso-shaped witness to its non-termination. Most of the state-of-the-art methods are incapable of proving non-termination of this program.

The black-box learning-based algorithm we propose can handle this program easily. This algorithm maintains a set of samples generated from the program, and makes progress by iteratively learning a candidate recurrent set (represented as a decision tree) consistent with the samples. The whole iterations of learning the recurrent set is depicted in Figure 4.



(a) initial (b) add negative $(0, 0)$ (c) add positive $(-1, 0)$ (d) add implicative $(-1, 0) \rightarrow (4, 0)$

Fig. 4: The iterations for proving non-termination of the motivating example. Positive and negative samples are depicted as dots (red=negative, blue=positive, green=unknown) and implicative samples are depicted as dashed lines. The learned candidate is shown in the upper right corner and shaded in the diagram.

In the beginning, the sample set is empty, and the decision tree learner (see Section II-B) returns `true` as the candidate. Obviously, $\varphi_2(\texttt{true})$, i.e, $\forall k, j.\ \texttt{true} \rightarrow k \neq 0$ is invalid, and suppose $(k = 0, j = 0)$ is the state returned by the SMT solver as counter-example. This state should be excluded from the next candidate. It is thus labeled as a *negative* sample and added to the sample set.

The learner is then invoked again with a sole negative sample $(k = 0, j = 0)$, and returns a new candidate `false`, which is invalid since it represents an empty state set. Intuitively, any state s that is reachable (i.e., with an initial state s' such that $(s', s) \models \mathcal{T}_{stem}$) and belongs to a recurrent set (i.e., $s \models R$) can be added as a *positive* sample at this point. However, this is not possible since the predicate R is still *unknown*, and we cannot determine a priori whether a state is contained in some recurrent set. Note that solving the constraint $\varphi_1(\texttt{false})$ cannot provide such a state, since φ_1 is an existentially quantified formula.

We tackle this problem by *speculating* a positive sample from the reachable state. Assume here we pick the state $(k = -1, j = 0)$. A selected state is called *spurious* if no recurrent set containing this sample exists; this is possible when all executions starting from this state terminates. Since it is unknown if the state is spurious at this point, the algorithm continues by regarding it as a genuine positive sample.

Now the learner is invoked again, with a negative sample $(k = 0, j = 0)$ and a positive sample $(k = -1, j = 0)$, and returns a predicate $k \leq -1$ as the candidate. This time, the validity check of $\varphi_3(k \leq -1)$, i.e. $\forall k, k', j, j'.\ k \leq -1 \wedge k' = -2 \times (k - 1) \times k \times k \wedge j' = 0 \rightarrow k' \leq -1$, would fail, since the candidate holds on the state $(k = -1, j = 0)$, but not on its successor state $(k = 4, j = 0)$. An *implicative* sample $((k = -1, j = 0), (k = 4, j = 0))$ is then added.

When the learner is invoked in the next iteration, $k \leq -1 \vee k \geq 4$ is inferred as the candidate. This is a valid recurrent set for the program, thus the teacher reports non-termination.

Discussions

As Figure 4 indicates, each sample generated steers the learner closer to a valid candidate. The learning process is efficient and converges quickly. Besides, since it is agnostic of the concrete program, the aforementioned difficulties concerning complex program constructs are not present.

However, the above learning iteration does not encounter spurious positive samples. What will happen if a wrong choice is made? Note that for a state s that is terminating, all its successors are also terminating. If s is chosen as a positive sample, by φ_3 of Definition 2, its successors are all contained in the recurrent set, including the final state that violates $\mathcal{G}_{loop}$, which should be negative. This entails that as the learning goes on, at some point the sample set must become inconsistent.

Suppose in the above discussion, instead of $(k = -1, j = 0)$ we choose $(k = 1, j = 0)$ as the positive sample. The learner would now return $k > 0$ as the candidate, and the validity check of $\varphi_3(k > 0)$ would again fail, leading to the implicative sample $((k = 1, j = 0), (k = 0, j = 0))$ being added to the sample set. The learner then fails to return any candidate, since the sample set is inconsistent.

In this scenario, we know the positive sample $(k = 1, j = 0)$ must be spurious[1] and lead to termination. To correct this, the teacher then flips this positive sample into negative, and speculates another state as the new positive sample, $(k = 1, j = 1)$ for instance. The details of how positive samples are speculated are documented in Section IV-B and Section IV-C, but in general, several iterations are needed before a spurious sample is corrected. To reduce the overhead, we expect as few spurious samples as possible.

At last, note that the decision tree learner requires a fixed set of attributes. It is possible that even if the speculated positive sample is not spurious, no recurrent set is expressible by the given attributes. To solve this problem, a special algorithm is needed to guarantee convergence of the learning, which is described in Section V.

IV. Black-Box Learning of Recurrent Sets

In this section, we introduce the black-box learning algorithm for learning a recurrent set. As is shown in Section III, the crux is to generate accurate positive samples and avoid inconsistency as much as possible. The learning algorithm is given in Algorithm 1. Given an input loop $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$, the algorithm either learns a valid

[1]Note this is different from black-box learning algorithm with an honest teacher, which generates no spurious sample and proceeds without backtracking. In that case, an inconsistent sample set directly entails that the target predicate does not exist.

recurrent set R and returns NT for non-termination, or returns T if termination is witnessed.

Algorithm 1: Black-Box Learning of Recurrent Sets

input : A loop $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$
output: (NT, R) if L is non-terminating, and R is a recurrent set of L; or T if termination is proved.

1 $S^+, S^-, S^\rightarrow \leftarrow \emptyset$; // $E \leftarrow \emptyset$, $m \leftarrow 0$
2 **while** *True* **do**
3 $R \leftarrow \texttt{Learner}(\texttt{S}^+, \texttt{S}^-, \texttt{S}^\rightarrow)$; // BoundedLearner ()
4 **if** $\exists s.\ s \models \neg(R(\vec{x}) \rightarrow \mathcal{G}_{loop}(\vec{x}))$ **then**
5 $S^- \leftarrow S^- \cup \{s\}$;
6 **else if** $\exists s, s'.\ (s, s') \models \neg(R(\vec{x}) \wedge \mathcal{T}_{loop}(\vec{x}, \vec{x}') \rightarrow R(\vec{x}'))$ **then**
7 $S^\rightarrow \leftarrow S^\rightarrow \cup \{(s, s')\}$;
8 **else if** $S^+ \neq \emptyset$ **then**
9 **return** (NT, R) ;
10 **while** $S^+ = \emptyset \vee \texttt{SampleConflict}(\texttt{S}^+, \texttt{S}^-, \texttt{S}^\rightarrow)$ **do**
11 $S^- \leftarrow S^- \cup \texttt{Generalize}(S^+)$;
12 $s_{pos} \leftarrow \texttt{SpeculatePositive}(S^-, S^\rightarrow)$; // SpeculatePositiveBounded ()
13 **if** $s_{pos} = \texttt{Nil}$ **then return** T ;
14 $S^+ \leftarrow \{s_{pos}\}$;

A. Overall Algorithm

The learning algorithm follows the black-box learning framework (Figure 1) which alternates between teacher and learner. The learner we use is the ICE decision tree learner (Section II-B), while Algorithm 1 demonstrates the teacher. A set of samples $S = (S^+, S^-, S^\rightarrow)$ is maintained, and initially all sets are empty (Line 1).

At each iteration, the learner is invoked to obtain a new candidate R (Line 3). The candidate R's validity is then checked (Line 4 to 9). If $\varphi_2(R)$ of Definition 2 is violated (Line 4), there must be a state s in the current candidate that does not satisfy $\mathcal{G}_{loop}$. The state s is then labeled as negative and added to S^-. Likewise, if $\varphi_3(R)$ is invalid (Line 6), the current candidate is not inductive, and we can find a state s that satisfies R, but one of its successors s' does not. The pair (s, s') is then added to the implicative samples $S^\rightarrow$. Otherwise, if the candidate passes all the checks, a valid recurrent set is found and NT is returned (Line 9).

When the validity checking fails, before invoking the learner again, the teacher checks if S^+ is empty, or the sample set becomes inconsistent after adding a new sample (Line 10). The subroutine SampleConflict checks the consistency of a sample set by computing the closure of S^+ and S^- under $S^\rightarrow$, and check if $S^+_\rightarrow \cap S^-_\rightarrow = \emptyset$. In either case, a fresh positive sample is needed. The teacher then tries to speculate a fresh one with the subroutine SpeculatePositive.

B. Speculated Positive Sample

A valid recurrent set R should satisfy φ_1 of Definition 2 as well. Given a candidate, this can be checked by an SMT solver, but since φ_1 is existential, no counterexample can be obtained if the check fails. We could only know that the current candidate is unreachable from any initial state. Besides, as is pointed out in Section I, dynamic execution can not produce a positive sample as well.

Instead of checking the validity of φ_1 and generating samples from the check, we choose to speculate positive sample directly so that if any candidate satisfies φ_2 and φ_3, it also satisfies φ_1. Intuitively, any state reachable via $\mathcal{T}_{stem}$ is a possible positive example, and a naive method is to check the formula $\exists \vec{x}'.\mathcal{T}_{stem}(\vec{x}', \vec{x})$ and choose any model of $\vec{x}$ as a positive sample. However, speculation inherently brings forth spurious positive samples and lead to inconsistency of the sample set. As a trade-off, backtracking of the positive samples (Line 10 to 14) is needed for soundness of the algorithm. To reduce the overhead, we strengthen the condition and try to generate positive samples as accurate as possible.

The subroutine SpeculatePositive is used to infer a fresh positive sample. It is invoked (Line 12) whenever the teacher finds the current sample set inconsistent. The positive sample speculated should be a state that:

- is reachable from an initial state,
- does not belong to the set of already known terminating states, and
- does not terminate within a fixed number of steps.

The reachability requirement is essential for φ_1, while the other two requirements are to help pick fewer terminating states. Formally, a state s is a speculated positive sample if it satisfies the following formula:

$$\rho(\vec{x}) := (\exists \vec{x}'.\ \mathcal{T}_{stem}(\vec{x}', \vec{x})) \wedge \neg S^-_\rightarrow(\vec{x}) \wedge \bigwedge_{i=0}^{k} \forall \vec{x}_0, \vec{x}_1, \ldots, \vec{x}_i. \left(\vec{x}_0 = \vec{x} \wedge \bigwedge_{j=0}^{i-1} \mathcal{T}_{loop}(\vec{x}_j, \vec{x}_{j+1}) \right) \rightarrow \mathcal{G}_{loop}(\vec{x}_i) \quad (1)$$

where $S^-_\rightarrow$ is the predicate representing the set of all known terminating states, i.e. the closure of S^- under $S^\rightarrow$ (defined in Section II-B). k is a predefined threshold, and basically in Equation (1) the loop is unrolled k times. It is required that any successor of $\vec{x}$ within k steps satisfies the guard condition $\mathcal{G}_{loop}$. The universal quantification may seem a problem for constraint solving, but since we assume (in Section II) the loop is deterministic, they can be simply eliminated by substitution.

If ρ is unsatisfiable, all reachable states have been proved to be terminating and T is returned (Line 13). Otherwise, a model s exists and it is chosen as the new positive example (Line 14). Since all states in $S^-_\rightarrow$ are blocked in ρ, the new positive sample must be a fresh one. While ρ can not exclude all spurious samples, it is effective for reducing the cost of backtracking when most terminating state terminates within a small number of steps.

C. Generalization of Spurious Positive Samples

Equation (1) enables efficient sampling of positive samples, but spurious positive samples are still inevitable during the learning. When this happens, we must prevent Equation (1) from sampling the same positive sample. This can be ensured by labeling the original positive sample as negative, i.e., updating S^- to $S^- \cup S^+$.

With the above trivial method, only a single state is blocked at a time. To improve the efficiency, in Algorithm 1 we generalize the single positive sample to a set of states that are guaranteed to be spurious (Line 11) before adding them to S^- and speculating a new positive sample. Intuitively, we try to find the reason for inconsistency of the sample set, represented as a predicate, and block all states that satisfy it. In this way, Equation (1) can be steered to obtain sample that's less likely to be spurious.

To generalize the positive sample, we adopt the interval counterexample of inductive invariant learning proposed in [24] to our setting. The idea is to generalize a sample to an interval by examining the unsat core of an unsatisfiable formula. Suppose s is the state represented by the spurious positive sample. Then starting from s, the loop must terminate after several, say m, iterations. Define

$$\psi := \vec{x}_0 = s \wedge \bigwedge_{i=0}^{m-1} \mathcal{T}_{loop}(\vec{x}_i, \vec{x}_{i+1}) \wedge \mathcal{G}_{loop}(\vec{x}_m) \quad (2)$$

Obviously, ψ is unsatisfiable.

Let $D = \{d_0, d_1, d_2, \ldots, d_{n-1}\}$ be a set of possible interval distances with $0 \in D$, where each d_i is an integer. We use the set D to strengthen the requirement on the initial state $\vec{x}_0$ and obtain the following formula that's also unsatisfiable [24]:

$$\psi' := \bigwedge\{x_0 \geq s(x_0) - d, x_0 \leq s(x_0) + d \mid x \in \vec{x},\ d \in D\} \wedge \bigwedge_{i=0}^{m-1} \mathcal{T}_{loop}(\vec{x}_i, \vec{x}_{i+1}) \wedge \mathcal{G}_{loop}(\vec{x}_m) \quad (3)$$

By making all the constraints $x_0 \geq s(x_0) - d$ and $x_0 \leq s(x_0) + d$ in ψ' *soft constraints* and invoking the SMT solver, we can obtain a subset of these constraints as the *unsat core*. Each of the *unsat core* constraints represents an interval $[s(x_0) - d, \infty)$ or $(-\infty, s(x_0) + d]$, and are more general than a single positive sample. By adding all states satisfying these constraints to S^-, we block a (possibly infinite) set of states from been considered as the positive samples.

Example 1. *Consider the motivating example in Figure 3. As discussed in Section III, the state $(k = 1, j = 0)$ is a spurious positive sample that leads to termination in one iteration.*

Suppose $D = \{0, 1\}$, then we generalize the sample by invoking SMT solver on the formula $k \geq 1 \wedge k \leq 1 \wedge k \geq 0 \wedge k \leq 2 \wedge j \geq 0 \wedge j \leq 0 \wedge j \geq -1 \wedge j \leq 1 \wedge k' = -2 \times (k-1) \times k \times k \wedge j' = 0 \wedge k' \neq 0$.

The solver returns $\{k \leq 1, k \geq 0\}$ as the unsat core, which represent exactly all the terminating states. Then by blocking $0 \leq k \leq 1$ and speculating any fresh positive sample, the correct recurrent set can be learned following the rest of the algorithm.

```
while (x ≠ -1) {
    if (x = 0) {
        x ← 1; y ← 2;
    } else if (2x = y) {
        y ← 0;
    } else if (y = 0) {
        x ← 2x; y ← 2x;
    } else if (x ≤ -2) {
        x ← x - 1;
    } else {
        x ← -1;
    }
}
```

Fig. 5: A non-terminating program where the non-terminating state $(x = 0, y = 0)$ is not contained in any recurrent set expressible by octagonal attributes.

D. Correctness

We show that the Algorithm 1 is correct:

Theorem 1. *Given any loop $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$ as input, if Algorithm 1 returns (NT, R), the loop does not terminate. Likewise, if the algorithm returns T, the loop terminates.*

Proof. When the algorithm returns (NT, R), the candidate recurrent set R passes the checks at Line 4, 6, and 8. The check at Line 8 ensures S^+ is non-empty, and it contains a speculated positive sample s_{pos}, which Equation (1) ensures to be reachable from an initial state. Thus, R satisfies all requirements of Definition 2 and is a valid recurrent set.

Algorithm 1 returns T only if no positive example can be picked from SpeculatePositive. From Equation (1), this ensures that every reachable state either has been proved terminating (i.e. it's in the set $S^-_{\rightarrow}$), or terminates within k steps. Thus, the loop terminates. □

V. Convergent Learning

Algorithm 1 suffers from a common problem of black-box learning: the learning algorithm itself is not guaranteed to terminate. Although Lemma 1 ensures the decision tree learner always terminates given a fixed set of samples, the overall iterative learning algorithm may not converge, since the set of candidate recurrent set is infinite.

In [10], the authors proposed a bounded learner that ensures convergence of learning. Basically, the learner requires that all constants c in the atomic predicates $a \leq c$ of a candidate are bounded by a threshold m, which is initially 0. m is only incremented when no valid candidate exists in the given threshold. Since the number of candidates satisfying the m-bounded requirement is finite, and each iteration of learning gives a different candidate, this bounded learner will eventually learn a valid candidate, if one exists.

Algorithm 2: Methods Used in Convergent Learning

```
1  Subroutine BoundedLearner():
2      R, failed ← BoundedICELearner(S+, S−, S→, m) ;
3      while failed do
4          E ← E ∪ S+;
5          s_pos ← SpeculatePositiveBounded() ;
6          S+ ← {s_pos} ;
7          R, failed ←
             BoundedICELearner(S+, S−, S→, m) ;
8      return R;
9  Subroutine SpeculatePositiveBounded():
10     while ρ'(x⃗) is unsatisfiable do
11         E ← ∅ ;
12         m ← m + 1 ;
13     Let s be a model of ρ'(x⃗);
14     return s;
```

However, this bounded learner cannot be directly adopted to our method, since in our algorithm, divergence may result from not only the infinite space of candidates, but also wrong choice of positive samples. It is possible that a speculated positive sample is a non-terminating state, but no recurrent set expressible by the attributes A that contains this state exists.

The program in Figure 5 gives an example of this scenario. It does not terminate if the initial state satisfies $x \leq -2 \vee 2x = y \vee x = 0 \vee y = 0$. This predicate is also a valid recurrent set, but it is not expressible by the attributes $A = \{x, y, \pm x \pm y\}$ (i.e. the octagonal domain). However, another recurrent set that can be expressed by A, namely $x \leq -2$, does exist. Generally, valid recurrent sets for a program are not unique, and a positive sample may belong to several of them. Whether a valid recurrent set can be found by Algorithm 1 depends on the speculated positive sample: if it is not spurious, Algorithm 1 never generates a new one. Thus, when the sample does not belong to any expressible recurrent set, the algorithm would fail.

For example, suppose Algorithm 1 chooses the non-terminating state $(x = 0, y = 0)$ as the positive sample for Figure 5. It will then diverge since the execution starting from this state is infinite, but not contained in any expressible recurrent set.

To address this problem, we propose a convergent learning algorithm that is guaranteed to terminate and learn a recurrent set, provided that a recurrent set expressible as the boolean combination of atomic predicates $a \leq c$ exists. Essentially, the algorithm bounds not only the maximum value of constants in the learner, but also the speculated positive sample (by some appropriate metrics) in the teacher.

The convergent learning algorithm follows the same general framework of Algorithm 1, and the differences are marked as blue comments in Algorithm 1. An extra set of states E, and a threshold m are maintained (Line 1). The main difference from Algorithm 1 is how the threshold m is used to bound the learner (Line 3) and the speculation of positive samples (Line 12).

The two subroutines used are shown in Algorithm 2. `BoundedLearner` is a wrapper around the ICE bounded learner, which tries to generate a candidate with constants bounded by m, i.e. $|c| \leq m$. Though we ensure the sample set is always consistent (Line 11 of Algorithm 1), the learner might still fail because no such candidate exists. In this case, the learner put the current positive sample into the set E, and invoke `SpeculatePositiveBounded` to generate another positive sample. This is done repeatedly until a candidate is learned.

The subroutine `SpeculatePositiveBounded` now only returns those state s with $\|s\| \leq m$, where $\|\cdot\|$ is any norm of a vector. It also blocks all states in the set E, which records states that have been chosen but not proved to be terminating yet. More specifically, the formula $\rho'(\vec{x})$ in Equation (4) is used to generate a positive sample. In case all states with $\|s\| \leq m$ are already considered or blocked, E is emptied and m is increased by 1 (Line 12).

$$\rho'(\vec{x}) := (\exists \vec{x}'. \mathcal{T}_{stem}(\vec{x}', \vec{x})) \wedge \neg S^{-}_{\rightarrow}(\vec{x}) \wedge \neg E(\vec{x}) \wedge \|\vec{x}\| \leq m \wedge \bigwedge_{i=0}^{k} \forall \vec{x}_0, \vec{x}_1, \ldots, \vec{x}_i. \left(\vec{x}_0 = \vec{x} \wedge \bigwedge_{j=0}^{i-1} \mathcal{T}_{loop}(\vec{x}_j, \vec{x}_{j+1}) \right) \rightarrow \mathcal{G}_{loop}(\vec{x}_i) \tag{4}$$

We now prove that the bounded learning algorithm always converges provided that an expressible recurrent set exists:

Theorem 2 (Convergence of Algorithm 2)**.** *Fixing a set of attributes $A = \{a_1, \ldots, a_n\}$, for any loop $L = (\mathcal{T}_{stem}, \mathcal{G}_{loop}, \mathcal{T}_{loop})$, the bounded learnign algorithm is guaranteed to terminate and returns a valid recurrent set, if L admits a recurrent set expressible as the boolean combination of predicates of the form $a \leq c$, where $a \in A$ and c is any integer.*

Proof. Suppose R is the valid recurrent set for L that can be represented as boolean combinations of $a \leq c$. Let c_0 be the maximum of all such constants c in R. From φ_1 of the definition of a recurrent set, there exist states s such that $\exists s'. \mathcal{T}_{stem}(s', s) \wedge R(s)$. Let s_0 be one such state with the smallest metrics $\|s_0\|$. Note s_0 exists because the norm is always non-negative.

Now let $m_0 = max(c_0, \|s_0\|)$. Since the learner only uses constants bounded by m, and the speculated positive sample is bounded by m as well, R is only learned when the threshold $m \geq m_0$. Besides, Algorithm 2 only increases m when all states s with $\|s\| \leq m$ has been tried and no recurrent set with constants bounded by m exists. Then R must be learned when $m = m_0$.

Moreover, the number of semantically-distinct predicate representable as boolean combination of $a \leq c$, where c is bounded by m, is finite. Whenever an invalid recurrent set

is learned, either a new sample is added by checking φ_2 and φ_3, or the current positive sample is labeled to negative. Thus, the candidates returned by a learner are all semantically distinct. Therefore, for each threshold m, only a finite number of iteration is needed. This entails that R can be learned in finite steps. Since Lemma 1 ensures the learner terminates, The learning algorithm is guaranteed to terminate. □

VI. Experiments

We have implemented the Algorithm 1 in a prototype tool called RSLearn. It is based on the Boogie program verifier [25] and its built-in Houdini invariant synthesis algorithm [26]. The teacher is implemented by leveraging the verification condition generator in Boogie and the learner is based on the original ICE-DT learner [10], on top of the classical C5.0 algorithm [23]. As a simple heuristic, we follow [10] and use a fixed sequence $2, 4, 8, \infty$ as the bound in counterexample generation, since the constants in a recurrent set tend to be small. We also implement the generalization method in Section IV-C with $D = \{0, 1, \ldots 10, 20\}$.

As is pointed out in Section II, closed recurrent set is not complete for programs with non-determinism in the loop. We under-approximate these programs before learning, by taking every non-deterministic assignment $x := \texttt{nondet}()$ as `skip`, i.e., keeping only x's original value. This simple heuristic helps handle trivial non-deterministic programs.

The attributes used for decision tree learning are configurable, and we use the octagonal domain (i.e. $\pm x \pm y$ for every pair of variables x, y in the program), plus the simple attribute x for each variable x. While using more complex attributes is possible and necessary for proving non-termination of certain programs, these attributes are sufficient for most cases in the benchmark.

In the subsections below, we evaluate RSLearn on public benchmarks. Since RSLearn takes a Boogie program as input, we translate the C benchmarks to Boogie beforehand. The parsing time difference is negligible. The experiments are all carried out on an Intel® Core™ i5-10400 @ 2.90GHz CPU with 16GB memory, and the timeout is set to 60 seconds. All tools are run in single thread.

A. Evaluation on Linear Programs

To evaluate the efficacy of RSLearn, we compare it with Ultimate [3], [27], a state-of-art program analysis suite that integrate several non-termination analysis methods like [28] into its trace abstraction framework, VeryMax [18], a conditional termination prover that utilize Max-SMT for proving termination and non-termination [6], and RevTerm, a recent non-termination analysis tool that reduces non-termination proving to invariant generation by program reversal [8] and has strong guarantee for relative completeness. To see the effectiveness of the generalization method in Section IV-C, we also evaluate RSLearn with generalization disabled. We denote this version RSLearnNG below.

We use the current benchmark from the category *C-Integer Programs* of the Termination and Complexity Competition (TermComp [29]). All programs in this benchmark are considered, including 111 non-terminating programs, 223 terminating programs, and one program whose termination is unknown (Collatz's Conjecture). These programs have only linear assignments, but several of them contain non-linear guard expression and complex control flows. While it is focused on proving non-termination, we run RSLearn on terminating cases to validate the soundness of Algorithm 1.

RevTerm is parameterized and requires specifying several configurations like template size and polynomial degree. We use the best configuration reported in the experiments of [8], obtained by running all configurations exhaustively. It thus varies for different cases. Since we are unable to retrieve the commercial solver Barcelogic [30], we use MathSAT5 instead. Other tools are run in their default settings.

The experimental results are shown in Table I. It demonstrates the number of cases solved by each tool, and the number of unique cases solved. We report the average and standard deviation of running time on all successfully solved cases by each tool, and on non-terminating cases. For RSLearn, we also report how many times the learner is invoked, which represents the number of iterations the learning takes.

Table I shows the efficacy of RSLearn on non-termination proving, as it solves more non-terminating cases than other tools, and is able to solve three cases that no other tools can. The three uniquely solved cases are all programs with non-linear guard expression, which is easily handled by our black-box learning algorithm. RSLearnNG solves one less case than RSLearn with a similar performance, and requires 47% more iterations to converge, showing the effectiveness of the generalization method.

The solved number by RevTerm is comparable to ours, but this requires running RevTerm on each configuration, and taking the best configuration for each case in the benchmark. As is pointed out in [8], this is not a practical and optimal strategy. VeryMax is the closest to our tool in terms of non-terminating cases solved, but the average time spent on non-terminating programs (10.32s) are much higher than that spent on all cases (3.62s).

In terms of performance, our tool is the most efficient, taking only 0.8s on average, which is 5x, 4.4x and 1.9x faster compared to Ultimate, VeryMax and RevTerm respectively. The average number of calls to the learner needed is 18 times, and in most cases, a recurrent set can be learned in a few steps. The scatter-plot in Figure 6 compares the time taken in more details. It can be seen that RSLearn runs faster than Ultimate and VeryMax on most of the cases. On simple cases where both tools can solve very efficiently (<1s), RSLearn is slightly slower than RevTerm, but RSLearn performs better on more difficult cases.

While Algorithm 1 can return `T` to indicate termination, RSLearn is not a tool designed for proving termination. It is able to prove 9 of the terminating cases, which are all simple ones where the natural loops in the programs are unreachable. Integrating a data-driven termination analysis algorithm into our tool would be considered a future work. Besides, no false

TABLE I: Evaluation of RSLEARN and other tools on the TermComp benchmarks. The rows NO/YES/UNKNOWN contains the total number of cases each tool proves non-terminating, terminating, or fails to give an answer. Unique NO contains the number of non-terminating cases only solved by this tool.

	ULTIMATE	VERYMAX	REVTERM	RSLEARNNG	RSLEARN
NO	98	103	101	107 (108)	108 (109[2])
YES	201	213	0	5	9
UNKNOWN	36	19	234	222	217
Unique NO	0	0	0	3	3
Average Time	4.07s	3.62s	1.54s	0.81s	0.87s
Standard Deviation	2.42s	7.22s	6.76s	2.77s	1.24s
Average Time (NO)	3.68s	10.32s	1.54s	0.83s	0.87s
Standard Deviation (NO)	2.65s	9.35s	6.76s	2.84s	1.28s
Average Calls to Learner	-	-	-	25.1	17.9

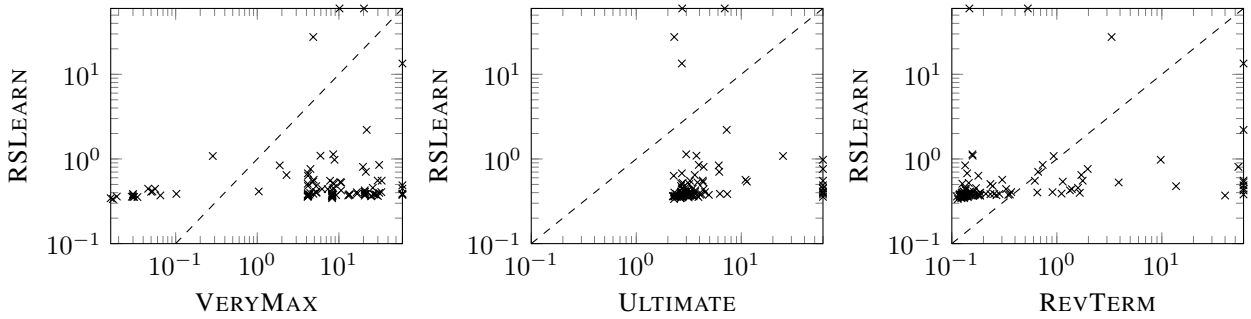


Fig. 6: Running time (s) of RSLEARN on non-terminating cases, compared with other tools. Cases that timeout count as 60s.

positive is found in the experiments, showing the soundness of the algorithm.

B. Evaluation on non-linear Programs

By following the black-box learning paradigm, our algorithm should be able to prove non-termination of non-linear programs efficiently. To show how RSLEARN performs on non-linear programs, we further evaluate our tool on the benchmark used to evaluate ANANT [7], which contains 29 non-linear non-terminating programs. This benchmark was later converted to C format and used for evaluating DYNAMITE [31]. Without using over-approximation, these programs are hard to deal with for typical white-box approaches.

We run RSLEARN along with VERYMAX, ULTIMATE and REVTERM on this set of non-linear programs and compare with their white-box approaches. The experimental results are shown in Table II. Since ULTIMATE is a practical termination analysis tool, it is able to solve 13 of the cases. On the other hand, RSLEARN is able to solve 28 out of 29 programs, and only fails a case (`p10.c`) due to non-determinism in the program, while VERYMAX and REVTERM struggles and solves only 2 and 0 cases, respectively. This shows the advantage of adopting black-box methods.

In terms of runtime performance, RSLEARN solves these cases very efficiently, and only take 12 iterations on average to synthesize a recurrent set. The average running time of RSLEARN is 0.46s, which is an order of magnitude faster than ULTIMATE and VERYMAX, and all cases are solved within one second.

We note that in [31], DYNAMITE is only able to solve 19 cases in 400 seconds. This shows the performance advantage of our static methods and learning-based approach, compared to DYNAMITE's approaches using dynamic execution. Besides, in [7] the tool ANANT, which the benchmarks are designed for, only solves 25 cases in 600 seconds, and RSLEARN runs much faster than ANANT on some cases with speedup up to 663x (`p18.c`). We believe this is due to the fact that ANANT must exhaustively extract lasso-shaped trace from the program, and then apply Farkas' Lemma for synthesizing a recurrent set, which does not scale well to the program size.

VII. RELATED WORK

Termination Analysis

A long line of work is focused on proving termination of programs, and we refer the reader to [32] for a summary of earlier termination proving techniques. In the last two decades, termination analysis remains an active research topic, and many practical tools emerge, including APROVE [1], ULTIMATE [3], HIPTNT+ [33] and 2LS [4].

The typical method for proving termination is to synthesize a ranking function by constraint solving [34]. This has been widely studied as complete methods for synthesizing different classes of ranking function, including linear [35], lexicographic [36], [37], and polynomial [38] ones exist.

[2]The program `Overflow.c` is specially crafted so that it does not terminate due to overflow. RSLEARN is based on Boogie which considers fixed-width integer, and other tools does not. Thus, their results on this case are different.

TABLE II: The results of running RSLEARN, VERYMAX, ULTIMATE and REVTERM on the benchmark of non-linear programs in [7]. The timeout is set to 60 seconds for all tools.

Benchmark	RSLEARN	VERYMAX	ULTIMATE	REVTERM
p1	0.36s	×	5.79s	×
p10	×	×	×	×
p11	0.45s	×	×	×
p12	0.71s	×	3.38s	×
p13	0.41s	×	×	×
p14	0.38s	×	×	×
p15	0.47s	×	×	×
p16	0.38s	×	3.22s	×
p17	0.36s	×	3.20s	×
p18	0.36s	×	2.76s	×
p19	0.38s	×	×	×
p2	0.46s	×	2.96s	×
p20	0.49s	×	×	×
p2a	0.49s	×	2.85s	×
p3	0.39s	×	×	×
p4	0.48s	×	×	×
p5	0.49s	×	×	×
p6	0.94s	×	2.80s	×
p7	1.05s	×	2.75s	×
p8	0.38s	4.15s	2.73s	×
p9	0.39s	2.36s	2.34s	×
pfactorial	0.40s	×	×	×
pinteger_log	0.36s	×	×	×
pinteger _log_mul	0.36s	×	×	×
plasso_ example1	0.36s	×	2.79s	×
plasso_ example2	0.46s	×	3.22s	×
plasso_ example3	0.44s	×	×	×
pnCr_ combination	0.36s	×	×	×
ppower	0.39s	×	×	×
Total Solved	28	2	13	0
Average	0.46s	3.26s	3.14s	-
Std. Deviation	0.16s	1.26s	0.85s	-

A recent trend of termination analysis is to make use of learning-based technique for synthesizing ranking functions. Most of these works follow the "guess-and-check" paradigm by first obtaining a likely termination argument, and then employing an off-the-shelf checker to validate it. Among these work, some are based on white-box methods, including template-based synthesis [39], [18]. Others utilize black-box methods. For example, [40] uses syntax-guided synthesis to generate candidate ranking function from a grammar. Other works adopt machine learning models such as linear regression [12], [**?**], decision tree [13], support vector machine [14], [15] and neuron network [16], [17].

Non-termination Analysis

Proving non-termination is studied as the counterpart of termination analysis. Since originally proposed in [5], the standard method of non-termination analysis is to find a recurrent set. In [21], the variant of closed recurrent set is proposed. Similar to loop invariant, recurrent set can be regarded as a predicate satisfying certain requirements. White-box synthesis techniques are thus widely used for proving non-termination, including [7], [6], [8], [41], [42]. Notably, a recent work [8] uses program reversal to reduce the problem to invariant generation. Geometric non-termination arguments [28] is a different type of nontermination argument, and can be synthesized by white-box method.

Only a few works have studied proving non-termination using "guess-and-check" paradigm. In [40] conjunctive recurrent sets are inferred and refined exhaustively using syntax-guided synthesis. [31] uses dynamic execution traces as samples, and generates a candidate by using dynamic invariant inference tools. Neither of these methods guarantees progress and convergence of the algorithm. As far as we know, there is no non-termination analysis method that is based on machine learning techniques.

Black-Box Methods in Verification

Apart from the aforementioned applications in termination analysis, black-box technique has been utilized mostly in safety verification, where it is particularly adopted for invariant synthesis. Daikon [43] pioneers this field by inferring conjunctive invariants from dynamic tests. Houdini [26] improves the algorithm and ensures the inductiveness of invariant.

Learning was introduced to verification explicitly in [44]. Later on, Angluin's L^* algorithm for exact learning [45] is applied to areas such as synthesizing assume-guarantee contracts [46], [47], [48], error traces [49], or even models for neuron networks [50].

There are abundant methods that can be adopted as a learner, including decision tree [10], [51], [24], support vector machine [52], [53], PAC learning [54], quadratic programming [12], and neuron networks [55], [56], etc. Implicative samples are introduced to learning in the ICE framework for invariant generation [9], which are further extended to Horn implicative samples [57].

VIII. CONCLUSION

In this paper, we propose a decision tree-based black-box algorithm for learning recurrent set and proving non-termination. The learning process is guided by speculated positive samples and samples generated from counterexamples. The algorithm is agnostic of the concrete program which allows synthesizing complex recurrent set and proving non-termination of wide classes of programs. We guarantee convergence of learning by a bounded learning algorithm where a threshold is used to bound both the teacher and learner. The method is implemented in the tool RSLEARN, and we observe its efficiency compared to state-of-the-art tools, as it solves more cases using fewer time, and is able to handle non-linear programs with ease. These results are promising as a first attempt to adopt black-box learning into non-termination analysis.

REFERENCES

[1] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann, "Proving Termination of Programs Automatically with AProVE," in *Automated Reasoning*, ser. Lecture Notes in Computer Science, S. Demri, D. Kapur, and C. Weidenbach, Eds. Cham: Springer International Publishing, 2014, pp. 184–191.

[2] D. Beyer and M. E. Keremoglu, "CPAchecker: A Tool for Configurable Software Verification," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer, 2011, pp. 184–190.

[3] M. Heizmann, J. Hoenicke, and A. Podelski, "Termination Analysis by Learning Terminating Programs," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 797–813.

[4] P. Schrammel and D. Kroening, "2LS for Program Analysis," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, Apr. 2016, pp. 905–907.

[5] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu, "Proving non-termination," *ACM SIGPLAN Notices*, vol. 43, no. 1, pp. 147–158, Jan. 2008.

[6] D. Larraz, K. Nimkar, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "Proving Non-termination Using Max-SMT," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 779–796.

[7] B. Cook, C. Fuhs, K. Nimkar, and P. O'Hearn, "Disproving termination with overapproximation," in *2014 Formal Methods in Computer-Aided Design (FMCAD)*. Lausanne, Switzerland: IEEE, Oct. 2014, pp. 67–74.

[8] K. Chatterjee, E. K. Goharshady, P. Novotný, and Đ. Žikelić, "Proving non-termination by program reversal," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 1033–1048.

[9] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A Robust Framework for Learning Invariants," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 69–87.

[10] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 499–512, Jan. 2016.

[11] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan, "DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 405–421.

[12] A. V. Nori and R. Sharma, "Termination proofs from tests," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, Aug. 2013, pp. 246–256.

[13] S. Kura, H. Unno, and I. Hasuo, "Decision Tree Learning in CEGIS-Based Termination Analysis," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 75–98.

[14] Y. Li, X. Sun, Y. Li, A. Turrini, and L. Zhang, "Synthesizing Nested Ranking Functions for Loop Programs via SVM," in *Formal Methods and Software Engineering*, ser. Lecture Notes in Computer Science, Y. Ait-Ameur and S. Qin, Eds. Cham: Springer International Publishing, 2019, pp. 438–454.

[15] Y. Yuan and Y. Li, "Ranking Function Detection via SVM: A More General Method," *IEEE Access*, vol. 7, pp. 9971–9979, 2019.

[16] M. Giacobbe, D. Kroening, and J. Parsert, "Neural Termination Analysis," *arXiv:2102.03824 [cs]*, Oct. 2021.

[17] W. Tan and Y. Li, "Synthesis of ranking functions via DNN," *Neural Computing and Applications*, vol. 33, no. 16, pp. 9939–9959, Aug. 2021.

[18] C. Borralleras, M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "Proving Termination Through Conditional Termination," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, A. Legay and T. Margaria, Eds. Berlin, Heidelberg: Springer, 2017, pp. 99–117.

[19] B. Cook, A. Podelski, and A. Rybalchenko, "Termination proofs for systems code," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 415–426, Jun. 2006.

[20] Z. Kincaid, T. Reps, and J. Cyphert, "Algebraic Program Analysis," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 46–83.

[21] H.-Y. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. O'Hearn, "Proving Nontermination via Safety," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer, 2014, pp. 156–171.

[22] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, Mar. 1986.

[23] ——, *C4.5: Programs for Machine Learning*, ser. The Morgan Kaufmann Series in Machine Learning. San Mateo, Calif: Morgan Kaufmann Publishers, 1993.

[24] R. Xu, F. He, and B.-Y. Wang, "Interval counterexamples for loop invariant learning," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 111–122.

[25] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A Modular Reusable Verifier for Object-Oriented Programs," in *Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Berlin, Heidelberg: Springer, 2006, pp. 364–387.

[26] C. Flanagan and K. R. M. Leino, "Houdini, an Annotation Assistant for ESC/Java," in *FME 2001: Formal Methods for Increasing Software Productivity*, J. N. Oliveira and P. Zave, Eds. Berlin, Heidelberg: Springer, 2001, pp. 500–517.

[27] Y.-F. Chen, M. Heizmann, O. Lengál, Y. Li, M.-H. Tsai, A. Turrini, and L. Zhang, "Advanced automata-based algorithms for program termination checking," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 135–150.

[28] J. Leike and M. Heizmann, "Geometric Nontermination Arguments," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 266–283.

[29] J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada, "The Termination and Complexity Competition," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds. Cham: Springer International Publishing, 2019, pp. 156–166.

[30] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "The Barcelogic SMT Solver," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Gupta and S. Malik, Eds. Berlin, Heidelberg: Springer, 2008, pp. 294–298.

[31] T. C. Le, T. Antonopoulos, P. Fathololumi, E. Koskinen, and T. Nguyen, "DynamiTe: Dynamic termination and non-termination proofs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 189:1–189:30, Nov. 2020.

[32] B. Cook, A. Podelski, and A. Rybalchenko, "Proving program termination," *Communications of the ACM*, vol. 54, no. 5, pp. 88–98, May 2011.

[33] T. C. Le, S. Qin, and W.-N. Chin, "Termination and non-termination specification inference," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, Jun. 2015, pp. 489–498.

[34] J. Leike, "Ranking Templates for Linear Loops," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer, 2014, pp. 172–186.

[35] A. Podelski and A. Rybalchenko, "A Complete Method for the Synthesis of Linear Ranking Functions," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds. Berlin, Heidelberg: Springer, 2004, pp. 239–251.

[36] A. R. Bradley, Z. Manna, and H. B. Sipma, "Linear Ranking with Reachability," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, K. Etessami and S. K. Rajamani, Eds. Berlin, Heidelberg: Springer, 2005, pp. 491–504.

[37] L. Gonnord, D. Monniaux, and G. Radanne, "Synthesis of ranking functions using extremal counterexamples," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 608–618, Jun. 2015.

[38] E. Neumann, J. Ouaknine, and J. Worrell, "On Ranking Function Synthesis and Termination for Polynomial Programs," p. 15, 2020.

[39] C. Urban, A. Gurfinkel, and T. Kahsai, "Synthesizing Ranking Functions from Bits and Pieces," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, M. Chechik and J.-F. Raskin, Eds. Berlin, Heidelberg: Springer, 2016, pp. 54–70.

[40] G. Fedyukovich, Y. Zhang, and A. Gupta, "Syntax-Guided Termination Analysis," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 124–143.

[41] M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl, "Automated Detection of Non-termination and NullPointerExceptions for Java Bytecode," in *Formal Verification of Object-Oriented Software*, ser. Lecture Notes in Computer Science, B. Beckert, F. Damiani, and D. Gurov, Eds. Berlin, Heidelberg: Springer, 2012, pp. 123–141.

[42] C. David, D. Kroening, and M. Lewis, "Unrestricted Termination and Non-termination Arguments for Bit-Vector Programs," in *Programming Languages and Systems*, J. Vitek, Ed. Berlin, Heidelberg: Springer, 2015, pp. 183–204.

[43] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *Proceedings of the 22nd International Conference on Software Engineering*, ser. ICSE '00. New York, NY, USA: Association for Computing Machinery, Jun. 2000, pp. 449–458.

[44] J. M. Cobleigh, D. Giannakopoulou, and C. S. PĂsĂreanu, "Learning Assumptions for Compositional Verification," in *Tools and Algorithms for the Construction and Analysis of Systems*, H. Garavel and J. Hatcliff, Eds. Berlin, Heidelberg: Springer, 2003, pp. 331–346.

[45] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

[46] R. Alur, P. Madhusudan, and W. Nam, "Symbolic Compositional Verification by Learning Assumptions," in *Computer Aided Verification*, K. Etessami and S. K. Rajamani, Eds. Berlin, Heidelberg: Springer, 2005, pp. 548–562.

[47] F. He, B.-Y. Wang, L. Yin, and L. Zhu, "Symbolic assume-guarantee reasoning through BDD learning," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, May 2014, pp. 1071–1082.

[48] F. He, S. Mao, and B.-Y. Wang, "Learning-Based Assume-Guarantee Regression Verification," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 310–328.

[49] M. Chapman, H. Chockler, P. Kesseli, D. Kroening, O. Strichman, and M. Tautschnig, "Learning the Language of Error," in *Automated Technology for Verification and Analysis*, ser. Lecture Notes in Computer Science, B. Finkbeiner, G. Pu, and L. Zhang, Eds. Cham: Springer International Publishing, 2015, pp. 114–130.

[50] Z. Xu, C. Wen, S. Qin, and M. He, "Extracting automata from neural networks using active learning," *PeerJ Computer Science*, vol. 7, p. e436, Apr. 2021.

[51] S. Krishna, C. Puhrsch, and T. Wies, "Learning Invariants using Decision Trees," *arXiv:1501.04725 [cs]*, Jan. 2015.

[52] J. Li, J. Sun, L. Li, Q. L. Le, and S.-W. Lin, "Automatic loop-invariant generation anc refinement through selective sampling," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 782–792.

[53] R. Sharma, A. V. Nori, and A. Aiken, "Interpolants as Classifiers," in *Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds. Berlin, Heidelberg: Springer, 2012, pp. 71–87.

[54] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori, "Verification as Learning Geometric Concepts," in *Static Analysis*, F. Logozzo and M. Fähndrich, Eds. Berlin, Heidelberg: Springer, 2013, pp. 388–411.

[55] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, "Learning Loop Invariants for Program Verification," in *Advances in Neural Information Processing Systems*, vol. 31. Curran Associates, Inc., 2018.

[56] J. Yao, G. Ryan, J. Wong, S. Jana, and R. Gu, "Learning nonlinear loop invariants with gated continuous logic networks," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 106–120.

[57] P. Ezudheen, D. Neider, D. D'Souza, P. Garg, and P. Madhusudan, "Horn-ICE learning for synthesizing invariants and contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 131:1–131:25, Oct. 2018.